
UNIVERSITÀ DI PISA

***Computer Engineering***

# Cloud Computing

# Word Co-Occurrence by Stripes Design Pattern

Academic Year: 2023/2024

*Francesco Taverna*

# Index

# Algorithm Word Co-Occurrence

The key idea is to count the number of times each pair of words $w_i$ and $w_j$ occurs in a collection of text files. For doing this, it has been developed a MapReduce program in Hadoop that computes word co-occurrences. Hadoop has been installed in *pseudo-distributed mode*.

## Stripes Design Pattern

Stripes is a design pattern utilized for matrix generation problem: given an input of size *N*, generate a squared output of size *N x N*. In this project the specific issue is to compute the co-occurrence matrix, thus, given a document collection, the algorithm will compute the number of times that a word $w_i$ co-occurs with a word $w_j$ and that number is represented by the cell $m_{ij}$ of the matrix.

Stripes has *O(N)* generation data in *O(N)* space.

The Stripes 'pseudo-code is the following:

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

1: class Reducer
2:     method Reduce(term w, stripes [H1, H2, H3, ...])
3:         Hf ← new AssociativeArray
4:         for all stripe H ∈ stripes [H1, H2, H3, ...] do
5:             Sum(Hf, H)                         ▷ Element-wise sum
6:         Emit(term w, stripe Hf)
```

*Map()* method takes a single record from a single document and creates an associative array for each term in the record. Each element of this array is a couple <String, Integer> and it's used to store the number of times in which a word *u* is in the neighborhood of *w*. Finally emits for each term *w* (the key), the array with all couples stored (the value).

*Reduce()* method has in input a term, that is the key, and all the associative arrays related to that key. As before, it declares another array in which will be stored the total number of occurrences of each word near that term *w* and it will be emitted at the end.

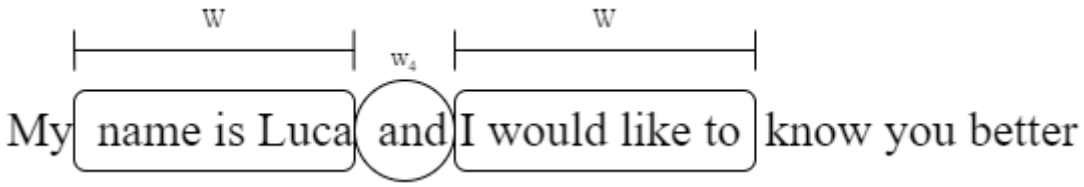
## Data used for Testing

The implemented algorithm has been run using a folder called “snippets” in which are contained 782 text files with name *lineXXX*, where *XXX* runs from 000 to 781. Each input file contains a text on multiple lines, including numbers, punctuation, etc.

# Implementation

A specific *map()* and *reduce()* functions have been defined to implement the pseudo-code previously described.

Concept of neighborhood chosen: given a word $w_i$, a term $w_j$ is considered a neighbor if it is included in a window that comprehends *W* words after and *W* words before $w_i$. *W* is a configuration parameter taken by *setup()* function in Mapper class for initializing *windowSize* variable.




In the case that the window exceeds the record, no other words are considered.

For implementation of associative arrays has been utilized the following HashMap type:

*Map<String, Integer>*

Final output record of *reduce()* function example:

*you {yielding=1, given=3, seed=21, bearing=11, for=13, tree=3, shall=8, call=2, it=42, have=11, to=1, every=1, herb=31, be=4}*

Punctuation is not considered in this algorithm, any sign is removed from record before emitting frequencies, unlike numbers, that are computed.

## Results

Hadoop spawns 782 map tasks, one for Input Split, that is a single text file in this case.

| windowSize | Number of Reducers | Bytes Written | Total time (minutes) | Mapper time (minutes) | Reducer time (minutes) |
|---|---|---|---|---|---|
| 3 | 1 | 18292804 | 61 | 60 | 50 |
| 3 | 2 | 18292804 | 63 | 62 | 51 |
| 3 | 3 | 18292804 | 65 | 64 | 54 |
| 6 | 1 | 27112594 | 62 | 61 | 50 |
| 6 | 2 | 27112594 | 61 | 60 | 49 |
| 6 | 3 | 27112594 | 61 | 60 | 49 |
| 12 | 1 | 33515352 | 62 | 61 | 50 |
| 12 | 2 | 33515352 | 61 | 60 | 50 |
| 12 | 3 | 33515352 | 61 | 60 | 49 |

Increasing the number of reducers tends to slightly increase the total execution time, this is seen with a window size of 3, the total time increases from 61 minutes with 1 reducer to 65 minutes with 3 reducers. This could be due to the overhead of coordinating more reducers: with a small input the overhead due to the usage of multiple reducers is more than the gain in balancing their load, instead, with a bigger input size having more reducers allows a balanced split of the workload, exploiting more parallelism. While setting window sizes of 6 and 12, the total time remains almost unchanged (61-62 minutes) regardless of the number of reducers, indicating that increasing it does not significantly impact job efficiency, but with more data to be processed doesn't get worse. Therefore, the increase in *windowSize* does not seem to significantly affect the total execution time, this probably can be addressed to the small input size. Indeed, doubling the window size over, for example, a double

number of text files could lead to an important increment of execution time (maintained fixed the number of reducers), since taking a wider window in a huge dataset has a strong impact on the quantity of bytes processed.
It is clear that increasing the window size from 3 to 12 results in a significant increase in bytes written. The mapping time is quite constant and almost equivalent to the total time, suggesting that the mapping phase is the main bottleneck, unlike the reducing one that lasts less. Thus, optimizing the mapping phase could lead to a significant reduction in the total execution time.

The progress of mapping during the execution is very slow and constant until the end and, at 18%, starts the reducing phase in parallel, even more slowly. A significant speed up can be seen at the end of mapping, when reducing pass from 33% to 100% in less than a minute, underlining how heavy mapping is, in terms of execution.

The execution times are big compared to the limited number of input files, this could be caused by the fact that the algorithm is executed in a pseudo-distributed mode, thus every node is run on a single machine, RAM and CPU are shared between Hadoop processes. In this mode, there is no distribution of map and reduce tasks between nodes.

## Conclusions

Increasing the number of reducers showed marginal improvements in processing time, hence maintaining a moderate number of reducers seems sufficient in this case, since increasing the number beyond a certain point does not offer significant benefits and can even increase the total execution time. The limited size of the input doesn't allow to exploit an increased number of reducers, because the overhead for managing the collaboration between them is more that the gain in terms of execution time. The impact of window size on processing time was minimal, suggesting that the algorithm is robust to variations in this parameter using that input. Further optimization and tuning, especially regarding the mapping phase, could potentially improve performance in scenarios with larger datasets or more complex processing requirements. Obviously, the number of bytes written is greater when the *windowSize* is bigger, in this case the neighborhood comprehends (almost) the entire row. The long execution time is probably due to the pseudo-distributed mode used for this testing, there is a single machine whose resources are shared between Hadoop processes.